A Coq Mechanization of JavaScript Regular Expression Semantics

NOÉ DE SANTO, AURÈLE BARRIÈRE, and CLÉMENT PIT-CLAUDEL, EPFL, Switzerland

We present an executable, proven-safe, faithful, and future-proof Coq mechanization of JavaScript regular expression (regex) matching, as specified by the latest published edition of ECMA-262 section 22.2. This is, to our knowledge, the first time that an industrial-strength regex language has been faithfully mechanized in an interactive theorem prover. We highlight interesting challenges that arose in the process (including issues of encoding, corner cases, and executability), and we document the steps that we took to ensure that the result is straightforwardly auditable and that our understanding of the specification aligns with existing implementations.

We demonstrate the usability and versatility of the mechanization through a broad collection of analyses, case studies, and experiments: we prove that JavaScript regex matching always terminates and is safe (no assertion failures); we identify subtle corner cases that led to mistakes in previous publications; we verify an optimization extracted from a state-of-the-art regex engine; we show that some classic properties described in automata textbooks and used in derivatives-based matchers do not hold in JavaScript regexes; and we demonstrate that the cost of updating the mechanization to account for changes in the original specification is reasonably low.

Our mechanization can be extracted to OCaml and JavaScript and linked with Unicode libraries to produce an executable regex engine that passes the relevant parts of the official Test262 conformance test suite.

CCS Concepts: • **Software and its engineering** → **Semantics**.

Additional Key Words and Phrases: ECMAScript, Regex, Mechanization, Coq

ACM Reference Format:
Noé De Santo, Aurèle Barrière, and Clément Pit-Claudel. 2024. A Coq Mechanization of JavaScript Regular Expression Semantics. *Proc. ACM Program. Lang.* 8, ICFP, Article 270 (August 2024), 29 pages. https://doi.org/10.1145/3674666

1 Introduction

Like most modern programming languages, JavaScript natively offers a way to write and match regular expressions (or *regexes*). These regexes are a convenient tool for text processing — so convenient in fact that a large proportion of JavaScript programs rely on them, including more than 30% of JavaScript npm packages [Davis et al. 2018, 2019]. The semantics of JavaScript is described in an open standard, ECMAScript [ECMA 2023], that specifies the behavior of the entire language with pseudocode.

The regex part of ECMAScript is surprisingly subtle and complex. This complexity comes from the presence of extended regex features not found in traditional regular expressions (*capture groups*, *backreferences*, *lookarounds*, *character classes*, *lazy quantifiers*, etc.; section 2), but also from some unconventional choices in the design of these features: even though other regex languages (like the ones in Python, Java, Go, Rust, Perl) share some of them, their semantics often differ in subtle

Authors' Contact Information: Noé De Santo, noe.desanto@epfl.ch; Aurèle Barrière, aurele.barriere@epfl.ch; Clément Pit-Claudel, clement.pit-claudel@epfl.ch, EPFL, Lausanne, Switzerland.

ACM 2475-1421/2024/8-ART270
https://doi.org/10.1145/3674666

ways [Davis et al. 2019]. For instance, JavaScript lookarounds are different from Java lookarounds, and JavaScript quantifiers and capture groups have unique semantics [Barrière and Pit-Claudel 2024]. To make matters worse, the ECMAScript regex standard is in continuous evolution: every recent revision of the standard contained significant additions or refactorings in the regex section, with dozens of individual edits every year; this churn caused Mozilla to abandon the development of its own regular expression engine and switch to Irregexp, an engine developed as part of the V8 JavaScript implementation used in Google Chrome and NodeJS, to better keep up with new features [Ireland 2020].

With all these features, JavaScript regex matching becomes significantly different from traditional regular expression recognition. First, the addition of backreferences changes the matching complexity from linear to NP-hard [Dominus 2000]. Second, the addition of capture groups changes the problem from recognition to segmentation: engines must not just compute whether a pattern matches a string, but also where the match occurred and which substrings were matched by each individual capture group (subexpressions in parentheses; this process is called *submatch extraction*). For instance, when matching the regex `a(b*)c` on the string "`abbcd`", an engine should return that there is a match, but also that the regex matched the four first characters "`abbc`" and that the capture group is defined and set to the value "`bb`". Supporting capture groups is very convenient, but it introduces ambiguity in the matching problem: when there are several ways to match a regex on a string, the language semantics must specify which way has the highest priority. In JavaScript, priority is given to the left branch of disjunctions. For instance, matching the regex `(a | ab)` on the string "`ab`" returns a result that only matches the substring "`a`" (this is in line with PCRE greedy match semantics, but not with POSIX leftmost longest match semantics). This disambiguation policy is particularly easy to implement — and to specify — using a backtracking algorithm.

For all the reasons above, the ECMAScript specification describes the JavaScript regex semantics with English pseudocode for a backtracking algorithm. There are multiple issues with this approach.

First, the pseudocode is not executable. It cannot be used directly to validate implementations of JavaScript, and it cannot be run to clarify ambiguities or validate proposed test cases. As a result, bugs continue to be found even in implementations of long-standing features. For example, until recently, one of the engines included in V8 contained bugs because of a misunderstanding of the semantics of the Kleene star quantifier [Barrière and Pit-Claudel 2024].

Second, the length and style of the specification make it unsuitable for formal reasoning. Mechanized semantics exist for substantial parts of the ECMAScript standard [Bodin et al. 2014; Guha and Lerner 2012; Maffeis et al. 2008; Park et al. 2015, 2020], but none of them support JavaScript regexes. As a result, previous work that tried to reason about JavaScript regexes had to redefine their own models of the semantics [Chida and Terauchi 2023; Loring et al. 2019]. Unfortunately, of the two published models that we are aware of, one is incomplete, and we found mistakes in both (section 2.4). To enable trustworthy reasoning about JavaScript regexes, one needs a faithful and auditable mechanized transcription of the ECMAScript pseudocode, without simplifications.

In that context, we present an **executable, proven-safe, faithful and future-proof Coq mechanization of JavaScript regexes**, as specified by the 2023 edition of ECMA-262 section 22.2. By *executable*, we mean that we can extract our mechanization either to OCaml or to JavaScript and run it. By *proven-safe*, we mean that we have a Coq proof that the compilation and matching process described in the specification terminates and that all assertions found in the specification are correct. By *faithful*, we mean that our mechanization follows the specification to the letter so as to be straightforwardly auditable (it is written in such a way that one can easily compare the pseudocode to the Coq definitions: we purposefully do not attempt to simplify the specification), and that it agrees with the spirit of the specification as understood by existing implementations. Finally, by *future-proof*, we mean that changes to the ECMAScript pseudocode require changes of comparable size to our mechanization, ensuring that it can be updated in the future as the standard evolves.

We claim the following contributions:

- **A shallow-embedded Coq mechanization** We show that ECMAScript regex semantics can be shallowly embedded in Coq, and we describe the encoding techniques required to represent exceptions, nonlocality, and possible nontermination in that style (section 3). We match the original specification as closely as possible: throughout the development, we interleave Coq code and English statements so that one can easily audit the mechanization. In total, we translated over 33 pages of pseudocode into Coq definitions.
- **Safety and termination proofs** Using our mechanization, we prove the correctness of several non-trivial properties of the ECMAScript regex semantics, demonstrating that our mechanization enables formal reasoning about JavaScript regexes. In section 4, we prove that the specification always terminates and that it cannot crash. In the process, we surface and mechanize crucial invariants.
- **A usable foundation for regex research** We show that our mechanization style is compatible with traditional program proofs and provides a usable formal foundation for regular expression compilation and optimization research by verifying a regex optimization excerpted from the Irregexp engine found in V8 (section 5).
- **An executable ground-truth for JavaScript regexes** We show that our mechanization style is compatible with extraction to OCaml and compilation to JavaScript. We link the result with Unicode libraries, and show that we can plug it into a JavaScript engine in a way that passes all relevant standard tests, albeit slowly in some cases (section 6.1). To the best of our knowledge, this is the first time that a candidate reference interpreter has been proposed for JavaScript regexes.

We start by giving a presentation of JavaScript regexes, their semantics, and related work formalizing them in section 2. We then describe our mechanization in section 3, highlighting the translation challenges that arise and the encoding techniques that make a shallow embedding possible and scalable (an error monad for failing operations of the specification, zipper contexts for nonlocal operations, and a proven-correct fuel encoding for general recursion). In section 4, we conjecture and formally verify a fundamental invariant of the ECMAScript regex specification, and we use it to prove that the specification always terminates and cannot fail. Then, in section 5, we present the formal verification of a regex transformation performed in Irregexp. Finally, in section 6, we present evidence that our mechanization is indeed executable, proven-safe, faithful, and future-proof. Specifically, we explain how to use extraction to generate two executable engines, one in OCaml and one in JavaScript. We show that the JavaScript engine can execute the Test262 official ECMAScript compliance test suite, and we evaluate on one example the effort required to extend our mechanization to support a new regex feature.

2 Understanding JavaScript Regexes

We begin this section with an overview of JavaScript regexes in section 2.1. We then describe their backtracking ECMAScript specification in section 2.2. In section 2.3, we give an informal explanation of a crucial property of all functions generated by the specification; we later mechanize this property to prove properties about the semantics (section 4). Finally, in section 2.4, we present some problems that we uncovered in previous formalizations of JavaScript regexes and illustrate the semantic subtleties that led to them.

Feature name	Explanation	Syntax
Empty regex		ε
Character		a, b, c, \\n, …
Character classes		[abc], [^A-Z], ·, \d, …
Concatenation		$r_1 r_2$
Disjunction (union)		$r_1 \mid r_2$
Greedy quantifiers	(a)	$r^*, r^+, r^?, r^{\{n,n\}}, \ldots$
Lazy quantifiers	(a)	$r^{*?}, r^{+?}, r^{??}, r^{\{n,n\}?}, \ldots$
Capturing groups	(b)	(r), $({}_{<id>}r)$
Non-capturing groups	(c)	$(?{:}\ r)$
Backreferences	(d)	\\$p$, \\$id$
Lookarounds	(e)	$(?{=}\ r), (?{\leq}\ r), (?{\neq}\ r), (?{\nleq}\ r)$
Anchors	(f)	^, \$, \b, \B

where n is a nonnegative integer, p is a positive integer, id is a named-group identifier (a string), r, r_1 and r_2 are regexes.

Fig. 1. Features of the ECMAScript regex language.

2.1 Features of JavaScript Regexes

The ECMAScript standard for regexes includes not only standard regular expression features (concatenation, disjunction, simple characters and Kleene quantifiers like * or $^+$), but also a number of extended features [ECMA 2023, Section 22.2]. Figure 1 summarizes the syntax of most of these features (we deviate from the plain-text JavaScript syntax in minor ways for readability purposes: for example, we write ε instead of nothing at all and $(?{\leq}\ r)$ instead of `(?<=r)`).

We now explain some of the extended features relevant to the remainder of this work, then contrast their semantics with those of the same features in other languages.

(a) **Quantifiers** specify that a regex should be matched repeatedly. $r^{\{n_1,n_2\}}$ means that the regex r should be matched between n_1 and n_2 times. If n_2 is not specified, then r can be matched an unbounded number of times. By definition $r^* = r^{\{0,\}}$, $r^+ = r^{\{1,\}}$, and $r^? = r^{\{0,1\}}$. *Greedy* quantifiers, like *, $^+$, and $^{\{n_1,n_2\}}$, give priority to matching the inner regex as many times as possible. *Lazy* quantifiers, like $^{*?}$, $^{+?}$, and $^{\{n_1,n_2\}?}$ give priority to matching the inner regex as few times as possible. For instance, $\mathtt{a}^+$ matches the first three letters of the string "aaab" while $\mathtt{a}^{+?}$ only matches the first one. To avoid infinite repetitions, the ECMAScript semantics forbids optional repetitions that match the empty string. For instance, when matching $()^+$, the semantics allows only one iteration of the plus. The first iteration is allowed (in fact, mandated) because the plus has one minimal mandatory repetition, but all further iterations are disallowed: they are optional, the body () of the quantifier matches only the empty string, and empty matches are not allowed for empty strings.

(b) **Capture groups** are used to record which part of the input string matched each paren-delimited sub-group of the original regex. When a regex matches a string, the engine must return not only the substring that was matched by the entire regex, but also all the substrings that were last matched by each capture group. For instance, consider the regex $\mathtt{a}^*(\mathtt{b}^*)(\mathtt{a}^*)$, with two capture groups, being matched on string "abbaaac". The expected result is that the regex matches the substring "abbaaa", that the first capture group $(\mathtt{b}^*)$ matches the substring "bb", and that the second capture group $(\mathtt{a}^*)$ matches the substring "aaa". Capture groups are numbered from 1 in order of appearance of their left parenthesis (this corresponds to a pre-order traversal of the AST). It is also possible to name capture groups (e.g. $\mathtt{a}^*({}_{<B>}\mathtt{b}^*)({}_{<A>}\mathtt{a}^*)$): this makes it possible to retrieve submatches by name and to refer to the group by name in backreferences.

(c) **Non-capturing groups** act like parentheses do in mathematics: they override (or make explicit) operator priorities. For instance, the * quantifier is applied to b in ab* but to ab in (?: ab)*.

(d) **Backreferences** match the text that was captured by an earlier group. For instance, the regex $({}_{<A>}\mathtt{a}^*)({}_{<B>}\mathtt{b}^*)$\A matches strings of the form $\mathtt{a}^n\mathtt{b}^m\mathtt{a}^n$ (two sequences of 'a's of the same length separated by a sequence of 'b's). Backreferences can also refer to unnamed groups by index. For instance, (a|b)\1 matches "aa" or "bb" but not "ab". Referring to a yet-undefined capture group (for instance for the regex \1(a)) is not an error: the backreference matches the empty string.

(e) **Lookarounds** condition matching on the surrounding context: they allow matching to proceed only if the preceding or following text matches another regex. For instance, the regex $(?\nleq \mathtt{c})({}_{<A>}\mathtt{a}^*)({}_{<B>}\mathtt{b}^*)$\A matches the same strings $\mathtt{a}^n\mathtt{b}^m\mathtt{a}^n$ as in the paragraph above, but only when not preceded by a 'c', because of the negative lookbehind $(?\nleq \mathtt{c})$. Similarly, with a positive lookbehind, $(?\leq \mathtt{b})({}_{<A>}\mathtt{a}^*)({}_{<B>}\mathtt{b}^*)$\A matches strings of the form $\mathtt{a}^n\mathtt{b}^m\mathtt{a}^n$ but only when preceded by a 'b' (lookarounds are *zero-width*, meaning that the initial 'b' is not part of the matched substring: the assertion only ensures that there is a 'b' before the match). Lookaheads instead condition the matching on the remainder of the string. For instance, a(?= b) will match the letter 'a' only when it is followed by a 'b'. Lookarounds include both lookaheads and lookbehinds, and can be either positive or negative.

(f) **Anchors**, sometimes also called *boundary assertions*, are similar to lookarounds: they condition the match on surrounding characters. ^ and $ assert that there are no characters before and after them respectively. \b asserts that either the character before it or after is a word character, but the other is not. Word characters are all characters treated as equivalent to letters and numbers by the Unicode specification [ECMA 2023, Section 22.2.2.9.2]. For instance, ^a does not find a match in the string "ba" since the 'a' is not at the beginning of the string.

ECMAScript flags. JavaScript regexes may be annotated with flags to configure the matching process. Examples include the i flag to make the matching process case-insensitive, the d flag to compute submatch positions, or the u flag to enable Unicode mode (we give more details about this mode in section 6.1).

2.1.1 *Comparison to Other Regex Languages.*

While similar features are found in most other regex languages, each language can make different choices of syntax and semantics [Davis et al. 2019]. These differences can be as simple as not using the same syntax: \A is an anchor in languages like Python and PCRE, but simply matches the character 'A' in JavaScript without the unicode flag.

But these differences can also be much more subtle. For instance, capture groups inside quantifiers have unique semantics in JavaScript [Barrière and Pit-Claudel 2024]. Each time a quantifier is entered, the values of all capture groups inside it are reset to `undefined` [ECMA 2023, Section 22.2.2.3.1]. This is specific to JavaScript regexes, and it implies, for instance, that matching the regex (?: (a) | (b))* on the string "ab" defines group 2 ((b)) but not group 1 ((a)): the first iteration of the star defines group 1, but upon entering the second iteration of the loop all groups defined within the loop are reset, and after the second iteration only 'b' is set. Similarly, quantifiers that can match the empty string behave differently in ECMAScript and in other regex languages [Barrière and Pit-Claudel 2024].

```
           MatchState  :=  (String × EndIndex × Captures)
          MatchResult  :=  MatchState  or  mismatch
 MatcherContinuation  :=  MatchState → MatchResult
              Matcher  :=  MatchState → MatcherContinuation → MatchResult
```

Fig. 2. Summary of the different types appearing in the compilation process.

2.2 ECMAScript Regex Matching

To understand our mechanization, we now present the inner workings of the pseudocode algorithm that specifies ECMAScript regex matching. The specification divides the process of matching a regex such as (a) | bc against an input string into four steps: parsing, validation ("early errors"), compilation, and execution:

(1) **Parsing** [ECMA 2023, Section 22.2.1] The textual representation of the regex is turned into an *Abstract Syntax Tree* (AST).
(2) **Early errors** [ECMA 2023, Section 22.2.1.1] Regexes with undesirable properties (called *early errors*) are rejected. Examples include:
   - $r^{\{n_1,n_2\}}$ when $n_1 > n_2$;
   - ($_{<G>}$a)($_{<G>}$b), which defines the group G twice;
   - \G, which has a dangling named backreference: the named group G is referred to but never defined.
(3) **Compilation** [ECMA 2023, Section 22.2.2] The AST of the regex is transformed into a matcher function which takes two arguments: an input string to search for a match, and an index (a position in the string) to start the search at. This function returns either `mismatch`[1], indicating that no match was found, or a `MatchState` [ECMA 2023, 22.2.2.1] containing the input string, the index at which the match ended, and a description of the capture groups defined during the matching process.
(4) **Execution** [ECMA 2023, Section 22.2.5] At a later point, when the user calls one of the JavaScript RegExp API functions (`match`, `matchAll`, `exec`, etc.), the generated matcher function is called through the `RegExp.prototype.exec` function.

Our mechanization covers the last three: we delegate parsing to a preexisting regex parsing library for reasons that we detail in section 6.1. Our mechanization represents regex objects using an inductive type, with each constructor corresponding to a production rule of the ECMAScript regex-parsing grammar.

2.2.1 ECMAScript Regex Compilation and Execution. Perhaps surprisingly, the paper specification of ECMAScript regular-expression matching does not define an interpreter or an inductive matching relation. Instead, the specification defines a compilation algorithm that transforms the parsed regex AST into a matching function, and separately explains how to execute the matching function on the input string.

Figure 2 summarizes the different types used by the regex specification. `MatchState` represents the internal state of the backtracking engine. It contains the original input string, the current position in that string (called `endIndex`, an integer), as well as the capture indices of each group that has already been defined (`Captures`, an array of indices). A `MatchResult` is either the `MatchState` obtained at the end of a match or the special value `mismatch` indicating that no match was found. A `MatcherContinuation` is a function from a `MatchState` to a `MatchResult`. Finally, a `Matcher` is a function taking a `MatchState` and a `MatcherContinuation` and returning a `MatchResult`.

[1]Named `failure` in the specification, but in this paper we prefer `mismatch` to avoid confusion with assertion failures.

The compilation is done by the function `compilePattern` [ECMA 2023, 22.2.2.2], with most of the work happening in the function `compileSubPattern` [ECMA 2023, 22.2.2.3]. The `Matcher`'s first argument is its initial state. The second argument is a continuation function used to continue the matching if this matcher succeeds (initially, this continuation is the identity function). During the execution of the matcher functions, this continuation function is used to encode regex concatenation.

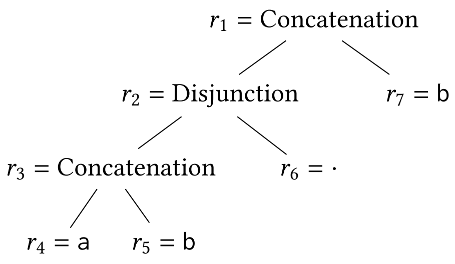


Fig. 3. Abstract syntax tree of $r_1 = (?{:}\ (?{:}\ \mathsf{ab}) \mid \cdot)\mathsf{b}$

Let us illustrate the backtracking behavior with an example. Consider matching the string "ab" with the regex $r_1 = (?{:}\ (?{:}\ \mathsf{ab}) \mid \cdot)\mathsf{b}$, whose AST is pictured in Figure 3. The dot · is a regex construct matching any single character. We write m_i for the matcher compiled from the regex r_i. A trace of the execution of the semantics is shown in Figure 4; we explain it here in more details.

Step 0 First, m_1 is called with a state `x0` at index 0 and a trivial continuation `c0` (the identity).

Step 1 $r1$ is a concatenation: as such, its compiled matcher first creates a new continuation `c7`. This continuation encodes the matching of the right side of the concatenation (r_7 = b). It uses the matcher of r_7 (`m7`) and the initial continuation `c0` it was called with. With this new definition, `m1` calls the matcher of its left-hand side (`m2`) with the new continuation `c7`.

Step 2 `m2` is a disjunction. This is called a *choice point*: the backtracking pseudocode must try the first alternative, and later maybe backtrack to try the other one. `m2` first calls the matcher of the left alternative (`m3`) with continuation `c7` to try to match r_3 = ab followed by r_7 = b.

Step 3 `m3` is again a concatenation, so `m4` is called with a new continuation (`c5`), as in step 1.

Steps 4–5 `m4` tries to consume an 'a' and succeeds. It advances the state, yielding `x1` at position 1, and calls its continuation `c5` with it. This continuation in turn invokes `m5 x1 c7`.

Steps 6–7 `m5` tries to consume a 'b' and succeeds. It advances the state, yielding `x2` at position 2, and calls its continuation `c7`, which in turn invokes `m7 x2 c0`. At this point, the engine successfully matched "ab" using the left-hand alternative of the disjunction. But now, it must match r_7 = b.

Step 8 `m7` tries to consume a 'b' but fails since the end of the input string has been reached. It hence returns `mismatch`.

Step 9: backtracking Since the call to `m3 x0 c7` returned `mismatch`, we return to the call of `m2`. It now attempts a match using `m6`, which corresponds to r_6, the right-hand alternative.

Steps 10-13 The process continues as before for a few steps. At the end, the state `x4` is returned, which indicates that a match was found.

The recursive structure of the generated matchers makes it so that their call stack acts as an implicit backtracking stack: every time a matcher has multiple ways to match a regex (for instance for a disjunction or a quantifier), it tries them one by one until one succeeds. In contrast, when there is only one way to match, the matcher calls its continuation in tail position.

2.3 Matcher Invariant

The details given above are enough to understand a key invariant of matcher functions generated by the specification, which we conjectured and proved in the course of our mechanization effort.

Step	Call stack	States and continuations	TC	Comment
0	`[]`	`x0` := ab (position 0)		Initialization
		`c0` := `fun s ⇒ s`		
1	`[m1 x0 c0]`	`c7` := `fun s ⇒ m7 s c0`	✓	Concatenation
2	`[m2 x0 c7]`			Choice point (disjunction)
3	`[m3 x0 c7, m2 x0 c7]`	`c5` := `fun s ⇒ m5 s c7`	✓	Concatenation
4	`[m4 x0 c5, m2 x0 c7]`	`x1` := $\underline{\text{a}}$b (position 1)	✓	Consumes “a”
5	`[c5 x1, m2 x0 c7]`		✓	
6	`[m5 x1 c7, m2 x0 c7]`	`x2` := $\underline{\text{ab}}$ (position 2)	✓	Consumes “b”
7	`[c2 x2, m2 x0 c7]`		✓	
8	`[m7 x2 c0, m2 x0 c7]`			Cannot consume “b”
	Match failed $\Longrightarrow$ Backtrack to last choice point			
9	`[m2 x0 c7]`		✓	
10	`[m6 x0 c7]`		✓	
11	`[c7 x3]`	`x3` := $\underline{\text{a}}$b (position 1)	✓	Consumes “a”
12	`[m7 x3 c0]`	`x4` := $\underline{\text{ab}}$ (position 2)	✓	Consumes “b”
13	`[c0 x4]`			Returns x4

A check mark in the “TC” column indicates that a call in tail position is performed.
The underlined part of a `MatchState` indicates the characters that have already been consumed.

Fig. 4. Matching process for r_1 on input string “ab”.

Matcher Invariant: any call to a `Matcher` function with a state `x` and a continuation `c` either returns a mismatch, or calls its continuation `c` on another state `y` such that the `endIndex` of `y` is greater than or equal to the one of `x` and that both `x` and `y` have the same input string.[2] Intuitively, this corresponds to the fact that the strings matched by the two halves of a concatenation do not overlap.

This invariant is not only critical to understand the compilation process and the generated matchers, but also to formally reason about them, both when proving rewrites and when proving semantic properties. We formalize and prove a generalization of it in section 4.

2.4 Semantic Subtleties Missed by Previous JavaScript Regex Formalizations

Ours is the first mechanization of the ECMAScript regex standard, but not the first attempt to reason about JavaScript regexes formally: paper-based models of ECMAScript regex matching have appeared in multiple past publications [Chida and Terauchi 2023; Loring et al. 2019]. These models are typically incomplete (omitting complex features such as lookbehinds [Loring et al. 2019]). More worryingly, our efforts to faithfully mechanize the specification revealed that these models were also incorrect. In this section, we illustrate some of the most subtle issues.

Rewriting greedy and lazy quantifiers. The models presented in Loring et al. [2019] and [Chida and Terauchi 2023] incorrectly handle greedy and lazy question mark operators, $^?$ and $^{??}$. They assume that $r^?$ can be equivalently rewritten to $(?{:}\ r \mid \varepsilon)$, and that $r^{??}$ can be rewritten to the regex $(?{:}\ \varepsilon \mid r)$, allowing for a smaller and cleaner language without $^?$ and $^{??}$.

[2]The direction of the inequality is reversed for lookbehinds, since they traverse the string backwards. We omit this case for now and generalize the invariant in section 4.3.

Unfortunately, both of these transformations are incorrect in general. In particular, a problem occurs when r can match the empty string while defining capture groups.

For greedy question marks like $r^?$, consider the regex $r = ()$, and compare the semantics of $r^?$ and $(?\!: r \mid \varepsilon)$ on the empty string. In the $r^?$ case, the specification (section 2.1) allows at most one optional iteration of r, but this iteration may not match the empty string. Hence, the correct behavior is to not match r at all (the quantifier $^?$ does 0 iterations). The final result simply matches the empty string and does not define any capture groups. In contrast, $(?\!: r \mid \varepsilon)$ allows r to match the empty string. Hence, the final result in this case does define the first capture group (the one contained in r).

For lazy question marks like $r^{??}$, consider the regex $r = (?\!=(\mathtt{a}))^{??}\mathtt{ab\backslash 1c}$ being matched on string "abac". A compliant engine would not find a match. Matching would proceed as follows: the engine would first skip the lazy quantifier (doing so, the first capture group would not be defined), then consume "ab", then match the empty string with the backreference (backreferences to undefined groups match the empty string), and finally attempt to match "c". Because the input is "abac", the last step would fail, and the engine would backtrack. Backtracking would lead the engine to attempt to match the body of the lazy question mark and define the capture group (a). Unfortunately, because lookaheads are zero-width, this would lead to an empty iteration of the quantifier, which the specification disallows. Having exhausted all backtracking opportunities, a correct engine would then return a match failure.

Rewriting the regex to $(?\!: \varepsilon \mid (?\!=(\mathtt{a})))\mathtt{ab\backslash 1c}$ would not preserve this subtle behavior: unlike the original regex, this one does match "abac". Taking the empty left branch ε would not find a match, since it would not define the capture group. The semantics then backtracks and takes the right branch of the disjunction (unlike previously, this would now be allowed). Doing so, the engine would go into the lookahead and set the first capture group to "a". With that, matching as a whole would (incorrectly) succeed.

Incorrect claims about infinite loops. It is sometimes mentioned [Sakuma et al. 2012] that JavaScript and RE2 have the same way to prevent infinite looping in a star matching the empty string, which is not true and can lead to different match results, for instance for the regex $(?\!: \mathtt{a}^?\mathtt{b}^{??})^*$ on string "ab" [Barrière and Pit-Claudel 2024].

Leveraging mechanized specifications to validate simplified semantics. Although we quickly found a counterexample for the first transformation, we initially believed that the second one was correct. We then tried to formally verify it using our mechanization, and this attempt led us to the counterexample above.

The official ECMAScript regex standard is verbose and complex: it is no surprise that previously published models of JavaScript regular expressions attempted to capture regex semantics in a more succinct and practical style [Chida and Terauchi 2023; Loring et al. 2019]. We too would prefer to reason about regexes in such a style, but it cannot be at the cost of correctness: to be trustworthy, formalizations of JavaScript regexes must follow the ECMAScript standard to the letter and avoid semantically incorrect simplifications.

We hope that our mechanization will provide foundations to *prove*, rather than posit, the correctness of more succinct, readable, and practical specifications for JavaScript regex semantics.

3 A Shallow-Embedded Coq Mechanization of JavaScript Regexes

In this section, we present our Coq mechanization of the ECMAScript standard for JavaScript regexes. We describe a collection of encoding and translation techniques that make a manual faithful shallow embedding possible in Coq, and scalable to the size of the specification (about 33 printed pages). Our main goal was to produce a mechanization as faithful and close as possible to the original pseudocode specification, and we use two key ingredients to ensure that.

Shallow embedding. First, we use a shallow embedding, where each function and type that appears in the pseudocode is directly translated to a similar function or type in the mechanization. This is the most natural and straightforward way to have an equivalent mechanization.[3]

Line-by-line auditability. Second, we interleave each line of our Coq mechanization with English statements taken directly from the corresponding pseudocode in the ECMAScript specification. We see two main benefits to this style. First, it eases the burden of auditing our mechanization to check that it performs the same operations as the ECMAScript standard. Figure 5 shows an example of how close the two are once we define adequate notations and encodings. Second, it gives us a measure of future-proofness: if the ECMAScript standard evolves (for instance, by adding new regex features), this style ensures that changes to our mechanization remain commensurate to the changes in the original specification.

We start by presenting our solution to encode the potentially failing operations used by the ECMAScript pseudocode as a pure Coq implementation in section 3.1. Some operations are even *nonlocal*: they refer to the parent or siblings of the current AST node: we propose an adequate zipper encoding in section 3.2. Another issue when defining Coq functions is that Coq enforces termination. In section 3.3, we show how we encoded the loops and recursive functions used in the standard with a fuel parameter.

Our monadic encoding, zipper, and fuel parameter are simple and time-tested solutions that ensure a low barrier of entry to our Coq mechanization. We converged on them after trying various other solutions, including encoding the termination and absence of failures with dependent types, and finally choosing the most straightforward encodings to update and reuse.

3.1 Faithfully Encoding Potentially Failing Pseudocode in Coq

In various places, the specification makes use of language features that are not available in Gallina, the programming language of Coq. The most common ones are *assertions* and other potentially *failing* operations. Assertions are used extensively in the specification, but there is no such feature in Gallina. One possible solution would be to ignore assertions entirely. In fact, the ECMAScript standard itself states that:

> "Such assertions are used to make explicit algorithmic invariants that would otherwise be implicit. Such assertions add no additional semantic requirements and hence need not be checked by an implementation. They are used simply to clarify algorithms." [ECMA 2023, Section 5.2].

Instead of ignoring assertions, our approach consists in encoding them within our Coq mechanization, using the Coq type system and an error monad. Assertions can be split into two categories:

- Type assertions: assertions of the form `Assert: x is a MatchState` (e.g. Figure 5 step `3.a`). For these, we use the type system of Gallina to ensure that the assertion holds.
- Other assertions: assertions such as `Assert: xe <= ye`. For these, we use an error monad.

This approach has multiple benefits. First, we keep the mechanization close to the pseudocode specification. Second, it allows us to encode other potentially failing operations of the specification, like accessing the arrays holding the capture information in each `MatchState`. Third, keeping assertions helps understand and reason formally about JavaScript regex behavior.

[3]Historical comments made by one of the original authors of the ECMAScript regex specification [TC39 2022] suggest that the backtracking pseudocode may have been derived from a lambda-calculus-based model. In that sense, mechanizing the specification in Coq may be returning it to its roots.

```
Disjunction :: Alternative | Disjunction
  1. Let m1 be CompileSubpattern of Alternative with arguments rer and direction.
  2. Let m2 be CompileSubpattern of Disjunction with arguments rer and direction.
  3. Return a new Matcher with parameters (x, c) that captures m1 and m2 and performs the following steps when called:
      a. Assert: x is a MatchState.
      b. Assert: c is a MatcherContinuation.
      c. Let r be m1(x, c).
      d. If r is not failure, return r.
      e. Return m2(x, c).
```

(a) A sample of the specification [ECMA 2023, Section 22.2.2.3].

```
(** >> Disjunction :: Alternative | Disjunction <<*)
| Disjunction r1 r2 ⇒
  (*>> 1. Let m1 be CompileSubpattern of Alternative with arguments rer and direction. <<*)
  let! m1 =<< compileSubPattern r1 (Disjunction_left r2 :: ctx) rer direction in
  (*>> 2. Let m2 be CompileSubpattern of Disjunction with arguments rer and direction. <<*)
  let! m2 =<< compileSubPattern r2 (Disjunction_right r1 :: ctx) rer direction in
  (*>> 3. Return a new Matcher with parameters (x, c) that captures m1 and m2 and performs the
       following steps when called: <<*)
  (fun (x: MatchState) (c: MatcherContinuation) ⇒
    (*>> a. Assert: x is a MatchState. <<*)
    (*>> b. Assert: c is a MatcherContinuation. <<*)
    (*>> c. Let r be m1(x, c). <<*)
    let! r =<< m1 x c in
    (*>> d. If r is not failure, return r. <<*)
    if r is not failure then r else
    (*>> e. Return m2(x, c). <<*)
    m2 x c): Matcher
```

(b) Our mechanization of the sample above.

Fig. 5. Comparison of the paper specification with our mechanization.

On the other hand, by adding assertions, we introduce potential failures where the paper specification claimed that assertions add no additional semantic requirements. Consequently, we go even further and prove, in Coq (section 4.2), that all assertions always hold and that other potentially failing operations never fail.

Proving safety with an error monad. A simple error monad, as shown on Figure 6, can faithfully encode assertions and other potentially failing operations of the specification. `Failure` represents assertion failures and other kinds of failing operations (e.g. out-of-bounds array accesses), and `OutOfFuel` is a special case of failure that we use when encoding potentially non-terminating functions (see section 3.3). The `bind` notation chains operations together and achieves a result that is visually very close to the pseudocode specification, as demonstrated in Figure 5. Note that the specification also sometimes uses non-exhaustive pattern matching (for instance, assuming that some match result is not a mismatch), which can be encoded with our `destruct!` notation (we later prove that the missing branches are indeed never taken).

Every function of the specification is then wrapped in the error monad, from the compilation phase (compilation can fail, for instance, if a backreference refers to an undefined group) to the

```
Inductive Result (S: Type) :=
| Success (s: S)
| Failure
| OutOfFuel.

Definition bind m f :=
  match m with
  | Success v ⇒ f v
  | Failure   ⇒ Failure
  | OutOfFuel ⇒ OutOfFuel
  end.
```

```
Notation "'let!' r '=<<' y 'in' z" :=
  (bind y (fun r ⇒ z)).

Notation "'assert!' b ';' z" :=
  (if (negb b) then Failure else z).

(* Non-exhaustive match *)
Notation "'destruct!' r '<-' y 'in' z" :=
  (match y with
   | r ⇒ z
   | _ ⇒ Failure
   end).
```

Fig. 6. A simple error monad and its notations.

execution phase (the generated matcher functions can fail, for instance, if they access the capture array of a `MatchState` out of bounds).

In summary, all assertions of the ECMAScript standard are encoded either with type annotations or within the error monad, and all functions that return a value of type `T` while including potentially failing operations are changed to instead return a value of type `Result T`.

```
  ii. Let cap be a copy of y's captures List.
      [...]
viii. Set cap[parenIndex + 1] to r.
```

Fig. 7. Modifying a list in the compilation of groups [ECMA 2023, 22.2.2.7].

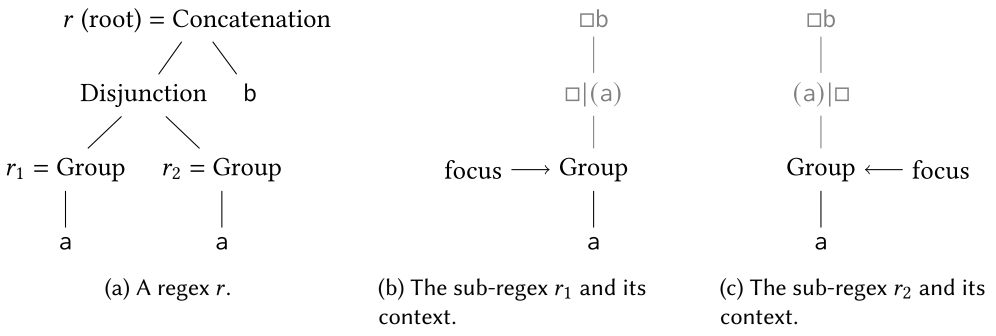
Avoiding mutation by recognizing specific usage patterns. Besides potential failures, the specification uses another imperative feature: mutable lists. In general, we would need a special encoding to handle mutable lists in a Coq mechanization. For instance, we could augment our monadic encoding with a state monad to hold the current value of each mutable list. However, the regex specification only uses mutable lists in very restricted cases, and they are always copied right before being modified. For example, when compiling capture groups [ECMA 2023, Section 22.2.2.6], a list is mutated at step `3.c.viii`, but this list is a local copy which was done at step `3.c.ii`. These two instructions are shown on Figure 7. Given this, we chose to only use functional immutable lists in our Coq mechanization. This keeps the mechanization simple to write and reason about while following the spirit of the standard.

3.2 Encoding Nonlocal Operations with a Zipper Context

An implicit feature of the AST representation of regexes used by the specification is that it is possible to navigate the AST upward, which in particular allows it to retrieve the *root* of the regex AST. This is used in a few functions of the specification. For example, as mentioned earlier, the index of a group is determined by counting the number of left parentheses appearing before it in the whole AST. This is done by the `CountLeftCapturingParensBefore` function [ECMA 2023, 22.2.1.3], shown in Figure 8. Consider the regex r = (?: (a) | (a))b defined in Figure 9. `CountLeftCapturingParensBefore` should return 0 when called on sub-regex r_1 (the first capture group), as no parentheses appear above or to the left of it. When called on r_2 (the second capture group), the function should return 1 as there is one opening parenthesis on its left, namely the one in r_1. This is one of the few functions in the ECMAScript regex pseudocode that are specified with such high-level descriptions, and without precise pseudocode instructions for each step.

1. Assert: *node* is an instance of a production in the RegExp Pattern grammar.
2. Let *pattern* be the *Pattern* containing *node*.
3. Return the number of *Atom* :: (*GroupSpecifier*$_{\text{opt}}$ *Disjunction*) Parse Nodes contained within *pattern* that either occur before *node* or contain *node*.

Fig. 8. Specification of the `CountLeftCapturingParensBefore` function [ECMA 2023, 22.2.1.3]



r (root) = Concatenation
Disjunction
b
r_1 = Group
r_2 = Group
a
a
(a) A regex r.
□b
□|(a)
focus ⟶ Group
a
(b) The sub-regex r_1 and its context.
□b
(a)|□
Group ⟵ focus
a
(c) The sub-regex r_2 and its context.

Fig. 9. A regex and its contextualized sub-regexes. Contexts are represented in gray.

This effectively means that regexes should include some form of context to relate them to the full regex they belong to, so that this function becomes feasible to implement. We model this pattern using zippers [Huet 1997]. The zipper holds both the current AST node and its *context*, allowing reconstruction of the root regex. Additionally, comparing the context of two regexes allows to distinguish syntactically identical sub-regexes that appear at different locations in the AST. Going back to the regex r, its child r_2 is represented by a pair of the AST for (a) and a context describing that this sub-regex is located on the left of the top-level concatenation, and then on the right of the disjunction: the context acts as a description of the path from the child to the root of the regex. Additionally, since we also want to be able to reconstruct the root regex, this path is augmented with all the data necessary to perform the reconstruction: contexts can be implemented with a linked list of incomplete AST nodes. These contexts are depicted on the right of Figure 9, where □ shows the incomplete parts of each sub-regex in their contexts. During compilation, as we traverse the AST of a regex, we manually maintain the associated context of each visited sub-regex.

3.3 Handling Arbitrary Recursion

The termination of most functions in the specification follows directly from the fact that these functions are structurally recursive on a regex. There are, however, notable exceptions: the `RepeatMatcher` function [ECMA 2023, 22.2.2.3.1] and some functions of sections 22.2.6 and 22.2.7 [ECMA 2023]. Since these are not structurally recursive, they are not accepted as-is by Coq, because it cannot ensure their termination.

There are multiple ways to encode non-terminating functions in Coq developments, for instance using coinductive definitions. We opted for a *fuel*-based solution: we added an extra fuel argument of type `nat` to these functions. If the function is called with a positive amount of fuel, then any subsequent recursive call is handed one unit less fuel; if the fuel is zero, then the function fails and returns `OutOfFuel`. From Coq's point of view, the function is then structurally recursive on this extra argument, and hence acceptable. Of course, this raises the question of whether there always exists an amount of fuel such that `OutOfFuel` is never returned. In section 4.2, we provide such a bound as a function of the regex and input-string sizes, and formally prove in Coq that this amount of fuel is sufficient to rule out all `OutOfFuel` failures. Given that termination can be proved, we find this to be a pleasant way to encode such non-terminating functions (alternative encodings such as coinductive functions are often perceived as much harder to reason about, and might force us to deviate further from the original pseudocode).

4 Deriving and Mechanizing New Semantic Properties: Validating the ECMAScript Specification

The ECMAScript specification itself could be incorrect, in the sense that its pseudocode could contain mistakes: for instance, it could produce functions that crash or do not terminate. In this section, we show that one crucial advantage of our mechanization is that we can formally verify semantic properties about the ECMAScript regex specification. In particular, we prove that, for any regex and any input string, the specification never crashes (e.g. assertions never fail and arrays are never accessed out of bounds) and always terminates. Although these are natural properties that one could expect from any regex specification, they can be quite difficult to infer just by looking at the pseudocode. With our mechanization, we can prove these properties in Coq, and as a result validate the correctness of the ECMAScript specification.

We originally proved these two properties separately. However, we later realized that both properties are direct consequences of the stronger matcher invariant that we informally described in section 2.3. We first present our mechanized matcher invariant and describe how it can be used to prove termination and absence of failures. We then present interesting cases of the proof that the matcher invariant holds for any matcher generated by the specification. Finally, in section 4.3, we explain how to generalize the simplified notion of progress used in this paper to the one found in the mechanization, which accounts for backwards input traversals performed by lookbehinds.

4.1 Mechanizing the Matcher Invariant

```
Definition matcher_invariant (m: Matcher) :=
(* For any valid state x, continuation c *)
∀ x c, Valid x →
  (* either there is no match or *)
  (m x c = Success mismatch) ∨
  (* m produced a valid state y which *)
  (∃ y, valid y ∧
    (* made progress with respect to x *)
    x ⩽ y ∧
    (* and was passed to c *)
    c y = m x c).
```

Fig. 10. Mechanized Matcher Invariant.

As explained in section 2.2, the matcher functions and the continuations compiled by the ECMAScript standard follow a particular control-flow, where each matcher function ends in a mismatch or in a call to its continuation. This call to the continuation is performed on a matcher state whose `endIndex` is greater than or equal to that of the state given as argument to the matcher, meaning that the matcher may have consumed some characters in the string before calling its continuation. In other words, every result of a matcher function call is either a mismatch or the result of calling its continuation after having progressed in the string. As it turns out, this invariant can be used to prove the termination and the absence of failures in the specification.

Once again, for presentation purposes we make the assumption that regexes only ever traverse the string forward, from its beginning to its end. This assumption is not true for lookbehinds (both positive and negative), which traverse the string backwards. We discuss how our approach can be generalized to both directions in section 4.3.

We first define the invariant in Coq as depicted in Figure 10. This invariant also captures the fact that matcher functions are only ever called on *valid* states, i.e. states whose `endIndex` is a valid

```
Theorem termination :
  (* For any regex r, string s *)
  ∀ r m s, compileSubPattern r = Success m ⟶
    earlyErrors r = OK ⟶
    (* the matcher cannot run out of fuel. *)
    m (init_state s) (identity_cont) ≠ OutOfFuel.
```

```
Theorem no_failure :
  (* For any regex r, string s *)
  ∀ r m s, compileSubPattern r = Success m ⟶
    earlyErrors r = OK ⟶
    (* the matcher cannot fail an assertion. *)
    m (init_state s) (identity_cont) ≠ Failure.
```

Fig. 12. Mechanized theorems for termination and absence of failures.

index of `input`. To encode this notion of progress in the string, we define a relation between match states, $\leqslant$, such that $x \leqslant y$ when x and y have the same input string and `endIndex(x)` $\leq$ `endIndex(y)`.

Proving the matcher invariant. With this definition, it is possible to prove that all matchers generated by compiling a regex satisfy the invariant. The corresponding theorem, which we proved in Coq, is shown in Figure 11. We explain the main challenges of the proof in section 4.2.

```
Theorem compiled_regex_invariant :
  (* For any compiled matcher m *)
  ∀ r m, compileSubPattern r = Success m ⟶
    earlyErrors r = OK ⟶
    (* the matcher invariant holds. *)
    matcher_invariant m.
```

Fig. 11. Proving the Matcher Invariant.

ECMAScript regex properties. Two important non-trivial properties of the ECMAScript regex semantics follow from the theorem of Figure 11: termination (functions generated by the specification always *terminate*, i.e. they never run out of fuel), and safety (the functions *cannot fail*, e.g. assertions always hold, and arrays and strings are never accessed out-of-bounds).

We state and prove the two theorems of Figure 12, where `init_state` is an initial matcher state containing the string to match and starting at index 0, and `identity_cont` is the continuation that always returns a success. The two proofs are immediate consequences of the `compiled_regex_invariant` theorem by contradiction: if the call to a matcher resulted in a lack of fuel or a failure, then because this result is not a `mismatch`, the continuation `identity_cont` itself would result in a lack of fuel or a failure, which is a contradiction.

Both termination and absence of failures are properties that one would expect about the specification, but that one cannot check formally without a mechanization. These proofs demonstrate that our mechanization enables formal reasoning about the JavaScript regex semantics in a proof assistant.

4.2 Proving Termination and Safety

Proving termination and safety (absence of failures) then amounts to proving the theorem of Figure 11. This theorem can be proved by induction on regexes. In this section, we describe some of the interesting cases.

Termination of the RepeatMatcher. To compile quantifiers, the ECMAScript standard defines a function `RepeatMatcher`, which takes as argument a matcher function for the inner regex, and returns a matcher function for the quantified regex. We show in Figure 13 a simplified version of our mechanization of this `RepeatMatcher` function, for the special case of matching the quantifier $e^{\{\mathtt{min,}\}}$, where `m` is the matcher function for e. In essence, this function computes a new continuation `d`, which repeats the matcher `m` as many times as possible and at least `min` times. The first `min` repetitions are allowed to match the empty string, but subsequent repetitions are required to make

```
Definition RepeatMatcher (m: Matcher) (min: nat) (x: MatchState) (c: MatcherContinuation) (fuel: nat) :=
  match fuel with
  | 0 ⇒ out_of_fuel
  | S fuel' ⇒
    let d := fun (y: MatchState) ⇒
      if min = 0 and endIndex(y) = endIndex(x) then mismatch
      else
      let nextmin := if min = 0 then 0 else min - 1 in
      RepeatMatcher m nextmin y c fuel'
    in
    if min ≠ 0 then m x d
    else
    let z := m x d in
    if z is not mismatch then z
    else c x
  end.
```

Fig. 13. Simplified pseudocode of RepeatMatcher [ECMA 2023, 22.2.2.3.1].

```
Lemma repeat_matcher_terminates :
  ∀ m x c min,
    (* for any continuation c that terminates *)
    (∀ y, c y ≠ OutOfFuel) →
    (* for any matcher m with the invariant *)
    matcher_invariant m →
    (* the matcher constructed by RepeatMatcher terminates *)
    RepeatMatcher m min x c (min+remainingChars(x)+1) ≠ OutOfFuel.
```

Fig. 14. Mechanizing termination of the RepeatMatcher.

some progress in the string; otherwise endIndex(y) = endIndex(x) and the continuation returns without performing any additional recursive calls.

Recall (from section 3.3) that we used fuel to define this function in Coq, as the function is not structurally recursive. Whenever we call this function, we call it with an extra argument, its fuel. To be able to prove by induction on the regex the theorem of Figure 11, in the particular case where the regex is a quantifier, we have to prove as an intermediate theorem that we can compute an initial fuel value to give to the RepeatMatcher such that it does not run out of fuel. We then state and prove the intermediate lemma described in Figure 14, using the bound $\text{min} + \text{remainingChars}(x) + 1$ as initial fuel, where $\text{remainingChars}(x)$ is the number of characters in the string which were not yet consumed in the match, i.e. $\text{remainingChars}(x) = \text{length}(\text{input}(x)) - \text{endIndex}(x) \geq 0$.

Informally, this bound comes from the fact that each time the RepeatMatcher iterates, either the value of min has decreased, or the matcher has made some progress in the string (because the matcher invariant holds for m). In the latter case, either m has not matched any character, and the RepeatMatcher immediately terminates, or m has matched some characters and the number of remaining characters decreases.

4.2.1 *Absence of Failures.*

When proving that the matcher invariant holds for any matcher function generated by the specification, we also need to prove that each assertion holds. We highlight here two interesting cases.

- Surprisingly, the notion of progress in the matcher invariant is not only helpful for the termination of `RepeatMatcher`, but is also useful to show that assertions cannot fail. For instance, this allows to prove that the assertion at step `3.c.vi` of the compilation of capture groups [ECMA 2023, Section 22.2.2.7] holds, since the assertion tests that progress has been made in the string.
- When compiling a named backreference, the compilation function first finds all named capturing groups using that name, then asserts that this list contains exactly one element.

```
(** >> AtomEscape :: k GroupName <<*)
| AtomEsc (AtomEscape.GroupEsc gn) ⇒
  (*>> 1. Let matchingGroupSpecifiers be GroupSpecifiersThatMatch(GroupName). <<*)
  let matchingGroupSpecifiers := groupSpecifiersThatMatch self ctx gn in
  (*>> 2. Assert: matchingGroupSpecifiers contains a single GroupSpecifier. <<*)
  assert! (List.length matchingGroupSpecifiers =? 1);
```

This assertion holds because the early-errors phase rules out regexes that define the same named capture group twice and regexes with a backreference to an undefined capture group. As a result, we proved that compilation always succeeds when the early errors phase succeeds, as follows:

```
Theorem early_errors_compile :
  (* For any regex without early errors *)
  ∀ r,  earlyErrors r = OK →
    (* compiling it to a matcher is a success *)
    ∃ m,  compileSubPattern r = Success m.
```

4.3 Generalizing Progress

Up to this point, our definition of the matcher invariant assumed that regex matcher functions always progress forward in the string. The proof we presented for the matcher invariant, from which both termination and absence of failures followed, crucially relied on this property.

However, JavaScript regexes include lookbehinds, which traverse the string backwards. In the actual specification, the compilation functions take an additional parameter indicating the direction of the compiled matcher, i.e. whether it should go through the input string forward or backward. This is notably used when compiling lookbehinds: when compiling $(?\leq r)$, r is compiled with the direction set to `backward`, regardless of the direction used when compiling the lookbehind. In our mechanization, we use a generalized notion of progress $\preccurlyeq$ that it is parameterized by a direction. In the context of the termination of `RepeatMatcher` (section 4.2), the definition of remainingChars(x) is also parametrized on the direction. Since the repeated matcher always goes in the same direction, our termination argument holds whether the matcher goes forward or backward.

5 Formally Verifying an Optimization Using our Mechanization

Mechanized semantics open the door to new research. In this section, we illustrate this claim with a case study showing that our mechanization can be used to formally verify a regex optimization used in a widely deployed JavaScript implementation, Irregexp.

As noted in section 2.4, subtleties of JavaScript regex semantics invalidate many natural and intuitive regex transformations that engines could use to simplify regexes. V8's Irregexp engine nonetheless performs regex rewriting during parsing: it replaces r^* with the empty regex ε whenever r can only match the empty string.[4] In this section, we build upon our Coq mechanization of the ECMAScript regex semantics to provide a Coq proof that this optimization is correct. Informally,

[4]https://github.com/v8/v8/blob/dd8d0b44d5d5b4bc3912026bf78f25ed3cafda1a/src/regexp/regexp-parser.cc#L3201

$$\frac{}{SN\ \varepsilon} \qquad \frac{}{SN\ \hat{}} \qquad \frac{}{SN\ \$} \qquad \frac{}{SN\ \backslash\mathrm{b}} \qquad \frac{}{SN\ \backslash\mathrm{B}}$$

$$\frac{}{SN\ (?\!=r)} \qquad \frac{}{SN\ (?\!\neq r)} \qquad \frac{}{SN\ (?\!\leq r)} \qquad \frac{}{SN\ (?\!\nleq r)}$$

$$\frac{SN\ r_1 \quad SN\ r_2}{SN\ r_1 \mid r_2} \qquad \frac{SN\ r_1 \quad SN\ r_2}{SN\ r_1 r_2}$$

$$\frac{SN\ r}{SN\ r^{\{n,\}}} \qquad \frac{SN\ r}{SN\ r^{\{n,m\}}} \qquad \frac{SN\ r}{SN\ r^{\{n,\}?}} \qquad \frac{SN\ r}{SN\ r^{\{n,m\}?}}$$

$$\frac{SN\ r}{SN\ (?\!:r)} \qquad \frac{SN\ r}{SN\ (r)} \qquad \frac{SN\ r}{SN\ ({}_{\text{<name>}}r)}$$

Fig. 15. Our strictly nullable analysis

```
Definition strictly_nullable_matcher (m: Matcher) :=
(* For any valid state, continuation c *)
∀ x c, Valid x ⟶
  (* either there is no match *)
  (m x c = mismatch) ∨
  (* or m produced a valid state y which *)
  (∃ y, valid y ∧
    (* did not make progress *)
    endIndex x = endIndex y ∧
    (* and was passed to c. *)
    c y = m x c).
```

Fig. 16. Strictly Nullable Matcher definition.

```
Theorem strictly_nullable_analysis_correct:
  ∀ (r: Regex) (m: Matcher),
    strictly_nullable r = true ⟶
    compileSubPattern r = Success m ⟶
    strictly_nullable_matcher m.
```

Fig. 17. Correctness of the *SN* analysis.

this transformation is valid because iterations of a star that do not consume any character in the string should be rejected by the star, meaning that the only acceptable behavior of the star is to perform zero iterations.

Our proof is conducted in four steps. We first define a syntax-directed analysis of regexes, identifying regexes that can only match the empty string, which we call *strictly nullable regexes.* This analysis is depicted on Figure 15, where *SN r* means that the regex *r* is strictly nullable. We include any anchors and lookaround. The definition is recursive for disjunction, concatenation, quantifiers and groups. Note that strictly nullable regexes can still look at the surrounding string with lookarounds or anchors; lookarounds can even be used to capture non-empty strings, despite not consuming any character from the input.

In our second step, we define a property of the matcher functions of such strictly nullable regexes. This property encodes the fact that such matcher functions cannot make any progress in the string. This can be seen as a stronger version of the matcher invariant mechanization of Figure 10, where the `endIndex` inequality has been replaced by an equality. This is defined on Figure 16.

In a third step, we can prove that every strictly nullable regex, according to the analysis of the first step, is compiled to a strictly nullable matcher according to our new definition. This theorem is shown in Figure 17. Its proof proceeds by induction on the regex. It closely resembles the proof of

```
Theorem strictly_nullable_same_matcher:
  ∀ (r:Regex) (mstar: Matcher) (mempty: Matcher),
    strictly_nullable r = true ⟶
    compileSubPattern (Quantified r (Greedy Star)) = Success mstar ⟶
    compileSubPattern Empty = Success mempty ⟶
     ∀ x c,  mstar x c = mempty x c.
```

Fig. 18. Correctness of the strictly nullable optimization

the theorem of Figure 11, except that we use the fact that the regex is strictly nullable to show that the intermediate state y on which the continuation is called has made no progress in the string.

Finally, we can conclude that it is correct to replace r^* with the empty regex when r is strictly nullable. We prove the theorem in Figure 18, showing that both r^* and the empty regex are compiled to equivalent matchers, returning the same results when called with the same inputs.

Even simple, seemingly intuitive transformations can be incorrect in JavaScript regexes (see section 2.4), and our Coq mechanization enables the formal verification of regex engine optimizations. One direction of possible future work could be to make the strictly nullable analysis even more precise. Currently, backreferences are never considered strictly nullable, and this is the case in both our analysis of Figure 15 and in the V8 optimization.[5] But when a backreference refers to a capture group whose body is itself strictly nullable, then the backreference itself should be strictly nullable. We believe that our analysis and proof could both be extended to justify even more aggressive optimizations.

6 Evaluation

We claim that our mechanization of ECMAScript regexes is proven-safe, executable, faithful and future-proof. Section 4 describes our mechanization: we have formally proved that the specification always terminates and cannot fail. We substantiate the claims of executability, faithfulness, and robustness to future changes in sections 6.1, 6.2, and 6.3. Finally, we discuss the formalization effort in section 6.4.

6.1 An Executable Mechanization

Using Coq's extraction mechanism [Letouzey 2008], we can generate OCaml code from our mechanization. This OCaml code can then be compiled, linked with adequate libraries, and executed to produce an independent engine for matching JavaScript regexes. Separately, to better integrate with the JavaScript ecosystem, we use Melange [Melange 2024] to translate the generated OCaml code to JavaScript.

As a result, from our Coq mechanization we can automatically generate two executable regex engines, one in OCaml, and one in JavaScript. Below, we highlight some interesting aspects of the extraction, compilation, and linking process.

Parsing. ECMA-262 specifies the syntax of regular expressions, but we did not port this part to Coq. Instead, we use an existing JavaScript regex parser library, `regexpp` [ESLint 2024] to parse regex strings into ASTs, and a small amount of code of JavaScript code to translate these ASTs into the representation used by our mechanization. This choice is motivated by a divergence within the ECMAScript standard, which defines two different grammars (one in section 22.2.1, and one in annex B.1.2). The former is the standard grammar, and the latter is the "legacy" grammar, intended

[5]https://github.com/v8/v8/blob/74530c3ff422d6e10ece2d9b68f0caec74c94a63/src/regexp/regexp-ast.h#L721

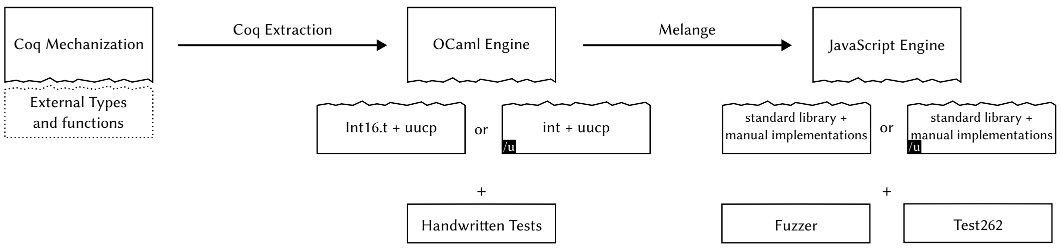


Fig. 19. Architecture of our development.

for web browsers but used by most implementations. These grammars do not agree on all inputs: for instance, the standard grammar rejects], whereas the legacy grammar allows it and treats it as a valid regex matching the character ‘]’. Given this, we left parsing out of our mechanization.

Unicode support. JavaScript regexes support Unicode through the u flag. In JavaScript, strings are represented as sequences of UTF-16 code units. The presence or absence of the u flag affects their tokenization during matching [ECMA 2023, Section 22.2.2]. Without the u flag, the input string is processed as-is, treating each code unit as one character. With the u flag, the input string is tokenized into a sequence of Unicode characters before processing. As an example, ‘🐢’ is a single Unicode character, U+1F422, but is encoded as a sequence of two code units (0xD83D 0xDC22) in UTF-16 (and hence in JavaScript strings). Matching this string against the regex · (which matches a single token) produces different results depending on the Unicode flag. When the flag is active, the regex matches the whole input, since the two code units get tokenized into just one Unicode character. Otherwise, the regex matches only the first code unit, 0xD83D: the engine considers that the string contains two distinct tokens.

In our mechanization, we reflect these tokenization choices by parameterizing the whole development on a type representing characters, along with functions to manipulate them (e.g., case foldings for case-insensitive matching). This means, in particular, that our specification does not include a specification of all Unicode: we delegate Unicode operations to external code that links against our specification.

We then provide two ways to instantiate the specifications, one corresponding to regexes with the u flag, and one without. The entire architecture of our development is explained in Figure 19, where both OCaml end JavaScript engines are completed with implementations of these types and functions: one for non-Unicode mode, and one for Unicode mode.

In the OCaml engine without Unicode mode, characters are encoded with the Int16.t type from the integers library. In the OCaml engine with Unicode mode, characters use the int type (large enough to represent all Unicode code points), and we use the OCaml library uucp to implement character manipulation functions like case foldings.

In the JavaScript engine, we use JavaScript strings to represent characters in all cases. We use standard library functions to implement some of the axiomatized character-manipulation operations (such as changing to uppercase), and manual implementations of the remaining ones (such as Unicode case foldings).

In other words: for case foldings or Unicode properties, the ECMAScript specification implicitly depends on an independent standard: Unicode. Our Coq mechanization abstracts the corresponding definitions away. To demonstrate the practicality of this approach, we instantiated most of these functions in our OCaml and JavaScript engines using existing Unicode libraries and manual implementations: we have support for all character manipulation functions, and we added support for one of the `\p` property classes as a sanity check.

6.2 A Faithful Mechanization

Our mechanization is trustworthy not just because we put great care into following the specification to the letter, but because we made every effort to validate it empirically. To do so, we used three layers of testing: we started with 95 custom tests, created by hand as we developed the mechanization, which we executed directly on the extracted OCaml code. Then, we compiled the extracted code to JavaScript, and made sure that the result passed the regex test suite used by most web browser vendors, Test262. Finally, to gain additional confidence, we subjected the same JavaScript code to hours of differential fuzzing against Irregexp, the regex engine found in V8.

Differential fuzzing. We implemented a differential fuzzer to cross-validate the correctness of our implementation with the Irregexp implementation from V8 [V8 2024], using randomly generated regexes and strings. The regexes and strings are generated on a small restricted alphabet of a few characters to improve the chances of finding matches. Our tests use randomly selected flags and start at random positions in the string. When comparing the results of our extracted engine and of Irregexp, we compare the value of the whole match and each capture group.

In this process, we directly generate regex ASTs, meaning that our fuzzer does not test the parser but rather the three phases of section 2.2 that we mechanized in Coq. Over thousands of such random tests, our extracted JavaScript regex engine has never disagreed with Irregexp.

Executing the Test262 suite. Test262 [TC39 2023] is the official ECMAScript test suite, covering all of JavaScript, and in particular regexes. To ensure that we faithfully mechanized the ECMAScript regex specification, we made sure that our extracted JavaScript engine passed the regex tests included in Test262.

Doing so required careful plumbing, because Test262 regex tests are not plain input-output pairs: they are complete JavaScript programs that call regex API functions and assert properties of the results. As such, these tests must be run with a full JavaScript engine: they cannot be run with a standalone regex engine. To validate our mechanization, we ran them with V8, but in a modified environment set up to route all calls to `RegExp.prototype.exec` through our own code. In this set up, any time a test calls `exec`, execution transfers to our code, and we can use `regexpp` to parse the given regex, forward the parsed regex and the input to our own JavaScript regex engine, and finally return the results to the caller, at which point normal V8 execution resumes using the result computed by our engine. This way, we can check that our mechanization agrees with the tests of Test262.

Relevant tests. Test262 contains 498 tests of core regex-matching functionality. They contain assertions like `assert.compareArray("abcdef".match(/(?<=ab(?=c)\wd)\w\w/), ["ef"], "#1");`[6], which checks the result of a call to `match` with nested lookarounds. These tests are directly relevant to our Coq mechanization. We pass 495 out of 498; the remaining 3 tests time out. The three tests that time out are character classes tests for the regexes \D, \W and \S which are checked exhaustively by testing all Unicode characters one by one. The algorithm described by the specification iterates over the entire character class for each input character, and since \D, \W, and \S include almost all Unicode characters, each test ends up executing billions of comparisons.

[6]https://github.com/tc39/test262/blob/c95cc6873d9933b8674ec8a840f43be0fb836ec1/test/built-ins/RegExp/lookBehind/nested-lookaround.js#L24

Other tests. Test262 also includes other tests that involve JavaScript regexes yet are not directly relevant to the parts of the semantics we mechanized in Coq. We detail them below:

- There are 445 prototype tests that check whether JavaScript objects related to regex matching have the right properties. For instance,

  ```
  assert.sameValue(RegExp.hasOwnProperty('prototype'), true, ...)
  ```
  [7]

  checks the presence of the `prototype` JavaScript property on the Regexp object. We can still run our extracted JavaScript regex engine on these tests, and we pass 436 out of 445 tests. Of the 9 tests that we fail, 5 are tests that fail because of V8.[8] The remaining 4 fail because of the JavaScript wrapper function that we use to intercept V8 and switch to our engine. For instance, one of the tests removes some string and array methods from the standard library and checks that regex matching still works[9], but our JavaScript wrapper uses some of these methods.
- There are 560 Unicode property-class tests that are mostly auto-generated [Bynens 2024] and test each individual character property class exhaustively. These are orthogonal to our mechanization: property classes are defined in a separate standard, the Unicode Character Database (UCD), and are handled by external Unicode libraries. Although we added one such property class as a sanity check, the corresponding tests would time out just like the ones for \D, \W, and \S above, so we excluded them.
- Finally, Test262 also contains 192 tests about experimental features currently under review for future inclusion in the specification. For example,

  ```
  assert.compareArray("bab".match(/(?<x>a)|(?<x>b)/), ["b", undefined, "b"]);
  ```
  [10]

  tests support for duplicate named groups, which are currently forbidden by the specification. As our mechanization is up to date with the current published specification, it does not pass these tests.

6.3 A Future-Proof Mechanization

The ECMAScript specification receives frequent updates, and so does the regex chapter. Our mechanization can easily be modified as the prose ECMAScript standard evolves. We claim that the changes required in our mechanization should be of the same order of magnitude as the changes to the specification itself.

As a case study, we delayed the implementation of word boundary assertions, or *anchors*, (^, $, \b and \B), to the very end of our mechanization effort, so we could simulate the addition of a simple feature to the specification. In this section, we document the changes that were needed to accommodate the introduction of anchors into the specification.

The first step is to extend the `Regex` type to add the four new constructors (one for each boundary assertion). Since these constructors are leaves of the AST, there is no need to extend the zipper with new context constructors.

Because we introduced new AST constructors, we are also required to update some methods of section 2.2.1 such as `CountLeftCapturingParensBefore` as well as `CompileSubPattern`, the main compilation function. Next, we need to implement the new cases in `CompileSubPattern` to compile

[7] https://github.com/tc39/test262/blob/c95cc6873d9933b8674ec8a840f43be0fb836ec1/test/built-ins/RegExp/prototype/S15.10.5.1_A1.js
[8] https://issues.chromium.org/issues/42203113 and https://issues.chromium.org/issues/42200389
[9] https://github.com/tc39/test262/blob/c95cc6873d9933b8674ec8a840f43be0fb836ec1/test/built-ins/RegExp/prototype/Symbol.replace/poisoned-stdlib.js#L10
[10] https://github.com/tc39/test262/blob/c95cc6873d9933b8674ec8a840f43be0fb836ec1/test/built-ins/RegExp/named-groups/duplicate-names-match.js#L11

```
(** >> Assertion :: ^ <<*)
| InputStart ⇒
    (*>> 1. Return a new Matcher with parameters (x, c) that captures rer and performs the
         following steps when called: <<*)
    (fun (x: MatchState) (c: MatcherContinuation) ⇒
      (*>> a. Assert: x is a MatchState. <<*)
      (*>> b. Assert: c is a MatcherContinuation. <<*)
      (*>> c. Let Input be x's input. <<*)
      let input := MatchState.input x in
      (*>> d. Let e be x's endIndex. <<*)
      let e := MatchState.endIndex x in
      (*>> e. If e = 0, or if rer.[[Multiline]] is true and the character Input[e - 1] is matched
           by LineTerminator, then <<*)
      if! (e =? 0)%Z ||!  (
           (RegExp.multiline rer is true) &&!
           (let! c =<< input[(e-1)%Z] in CharSet.contains CharSet.line_terminators c))
      then
        (*>> i. Return c(x). <<*)
        c x
      else
      (*>> f. Return failure. <<*)
      failure): Matcher
```

Fig. 20. Extension of `CompileSubPattern` for the \b construct.

boundary assertions. This process is made straightforward by our notations: most of the pseudocode can be transcribed almost literally, as shown in Figure 20. The most difficult line to translate is the one for case `1.e`. The condition requires indexing into the input string, which is an operation whose result is wrapped in the error monad as explained in section 3.1. As such, its result must be unwrapped using `let!` before passing it to `CharSet.contains`. We defined new boolean monadic notations `if!`, &&! and ||! to encode such potentially failing conditions.

The changes above are enough to fully support boundary assertions. We also adapted the proofs described in section 4 to these new constructs. As boundary assertions do not affect termination and do not have complex assertions, this only required a few additional lines of proof.

A similar process, albeit at a larger scale, should make it possible to support features added in the 2024 edition of the ECMAScript specification, or currently under discussion, such as the v flag[11], duplicated named groups[12], pattern modifiers[13], etc.

6.4 Formalization Effort

The ECMAScript specification contains around 33 pages dedicated to regex matching, which we mechanized in Coq. The table below summarizes our formalization effort. Additional handwritten tests were written alongside the mechanization.

[11] https://github.com/tc39/proposal-regexp-v-flag
[12] https://github.com/tc39/proposal-duplicate-named-capturing-groups
[13] https://github.com/tc39/proposal-regexp-modifiers

Language	Category	Lines of Code
Coq	Semantic mechanization	1810
Coq	Infrastructure and type definitions	788
Coq	Termination and safety proofs	1392
Coq	Strictly nullable optimization proofs	477
OCaml	Extracted engine infrastructure	688
OCaml	OCaml parameter implementation	244
JavaScript & OCaml	JavaScript parameter implementation	255
OCaml	Handwritten tests	973
JavaScript & OCaml	Differential fuzzer	374
JavaScript & OCaml	Test262 wrapper	137

7 Limitations and Future Work: Looking Ahead

We have formalized the ECMAScript regex matching process in its entirety, but not the external data that it depends on (Unicode property tables and character manipulation functions) nor the APIs that expose it.

In our mechanization, we have included as a comment, for each line of Coq code, the corresponding pseudocode line from the ECMAScript specification. To ensure completeness, we could build a tool to extract these comments and check that each and every line of the ECMAScript standard has been translated.

Formal verification of regex engines. As future work, we believe that our mechanized semantics would constitute the ideal specification for the correctness theorem of a formally verified JavaScript regex engine. Some JavaScript engines employ techniques fundamentally different from backtracking to avoid the particular cases where backtracking has exponential time complexity. For instance, both the Experimental engine in V8 [V8 2021] and recent work on linear matching of JavaScript regexes [Barrière and Pit-Claudel 2024] use a technique known as NFA simulation, which explores several paths in parallel. This approach is so different from backtracking that subtle semantic bugs have been uncovered [V8 2023]. Proving the correctness of this approach using our mechanization as a specification would provide high assurance that engines with linear-time complexity can be used to match JavaScript regexes.

Backtracking implementations sometimes use more advanced optimizations than the one we formally verified in section 5. For instance, Irregexp generates native code when the same regex is being executed several time [V8 2019], and recent work has explored the use of memoization to speed up backtracking engines even with modern extended regex features such as lookarounds or backreferences [Davis et al. 2021]. One could also use our mechanization to formally verify these techniques in Coq, for instance proving the correctness of a native code compiler for JavaScript regexes, or proving the correctness of an efficient memoized backtracking implementation.

Another kind of engine uses regex derivatives. Derivatives [Brzozowski 1964] present an elegant way to define the semantics of standard regular expressions. A recent addition to the .NET framework is a linear-time derivatives-based engine for a subset of .NET regexes (without backreferences or lookarounds) [Moseley et al. 2023]. To our knowledge, no derivatives-based engine exists for JavaScript regexes. One crucial step in defining a derivatives-based engine is to use regex rewriting rules to group together semantically equivalent regexes. In essence, for standard regular expressions, this ensures that the algorithm explores a *finite* equivalent automaton. A list of such rewriting rules for standard regular expressions (extended with regex intersection, complement and the empty-set regex) is available in [Owens et al. 2009, Section 4.1] and combining these rules has been proven to make the set of derivatives finite up to equivalence. However, for JavaScript regexes, we have

Derivative equivalence rules [Owens et al. 2009]

Rewriting Rule			JavaScript Counterexample
$r \mid r$	$\equiv$	r	$(?{:}\ (a) \mid (a))\backslash 2()\$$ on string “aa”.
$r_1 \mid r_2$	$\equiv$	$r_2 \mid r_1$	a | ab on string “ab”.

Rules from [Chida and Terauchi 2023; Loring et al. 2019] (see section 2.4)

Rewriting Rule			JavaScript Counterexample
$r^?$	$\equiv$	$r \mid \varepsilon$	$()^?$ on the empty string “”.
$r^{??}$	$\equiv$	$\varepsilon \mid r$	$(?{=}\ (a))^{??}$ab\1c on string “abac”.

Fig. 21. Counterexamples to standard regular expression rewriting rules we found during our mechanization.

already used our executable engine to show that several of these rules are incorrect in JavaScript, as shown on Figure 21. For the first rule, deleting branches may change the index of capture groups. For the second one, disjunction is notoriously non-commutative in JavaScript because the left branch has higher priority than the right one. This raises a new open question: is it possible to find a combination of rewriting rules leading to the same finiteness result for JavaScript regexes? If there is, we believe we could use our mechanization to formally verify each transformation, just like we did in section 5.

Formal verification of other JavaScript regex semantics. Previous work reasoning about JavaScript regexes [Chida and Terauchi 2023; Loring et al. 2019] have preferred a more denotational style of semantics, rather than mimicking the operational backtracking algorithm of the ECMAScript specification. We believe that these models could benefit from our mechanization by being proved equivalent to it in Coq. Similarly, the transducers semantics of Chen et al. [2022] have been tested against JavaScript, and with our mechanization we could prove that the two semantics are in fact equivalent.

Completing an existing JavaScript mechanization. Our mechanization could be used to complete the JSCert [Bodin et al. 2014] JavaScript mechanization in Coq. To do so, we would also need to mechanize some regex API functions that call our `RegExp.prototype.exec` function. As an experiment, we already mechanized the `search`, `test`, `match` and `matchAll` functions, and included these functions in our differential fuzzer. Our specification style seemed to work just as well for these functions as for the regex matching specification. We did not attempt to mechanize `replace` and `split`, because they call out to more generic functions about strings defined in other chapters of the ECMAScript specification.

8 Related Work

JavaScript mechanizations. Several works have tackled the mechanization of the ECMAScript standard. However, none of them covered the regex chapter. The common explanation is that the style used to define the regex semantics is quite different from the style used in other chapters [TC39 2022].

For instance, the JSCert project [Bodin et al. 2014; Charguéraud et al. 2018] manually translated most of the ECMAScript 5 specification into Coq definitions. Just like in our case, JSCert can then extract these Coq definitions to OCaml to obtain a reference interpreter, JSRef. JSCert models the complex control-flow of the ECMAScript semantics using pretty-big-step semantics [Charguéraud

2013], and also uses monadic operators to encode possibly failing operations. Our monadic definitions are more specialized to the kind of failures that occur in the regex chapter. An interesting direction for further work would consist in combining the JSCert development with our regex mechanization.

Instead of manually translating ECMAScript pseudocode, the ESMeta project uses JISET, an automatic translator, to transpile ECMAScript pseudocode into a custom intermediate representation [Park et al. 2020]. This allows the mechanization to be updated almost automatically when the ECMAScript standard evolves. This intermediate representation can then be used to define differential testers [Park et al. 2021b] and static analysers [Park et al. 2022, 2021a] for JavaScript. Despite JISET being able to translate more than 90% of the ECMAScript standard automatically, it also does not support the regex chapter (again, this is due to the peculiarities of the regex section). Unfortunately, it also does not yet support generation of definitions suitable for use in an interactive theorem prover such as Coq. To be able to rebuild our work on top of ESMeta, we would need to extend the JISET intermediate representation and translator to work with the regex specification style, and to add a translator from it to Coq; this could constitute interesting future work, and we hope that this paper would provide a good ground truth to evaluate against.

Regex formalizations. Other than related work formalizing JavaScript regexes [Chida and Terauchi 2023; Loring et al. 2019] (see section 2.4), there have been numerous formalizations of other regex languages, with various combinations of features. We present a selection of different semantic choices here, although the list is not exhaustive.

A formal model of a subset of .NET regexes has been presented in [Moseley et al. 2023], where the semantics is defined using derivatives extended with locations to support features like anchors and lookarounds. A similar approach has recently been mechanized in Lean [Zhuchko et al. 2024] for regexes with lookarounds and POSIX longest match semantics. Using derivatives to define the semantics of lookahead had been previously investigated in [Miyazaki and Minamide 2019]. In contrast with JavaScript regexes, these models do not include capture groups and backreferences.

There have also been different definitions of regex semantics using transducers, either *prioritized finite transducers* (PFTs) [Berglund et al. 2014; Berglund and van der Merwe 2017a] and later *prioritized streaming string transducers* (PSSTs) [Chen et al. 2022]. Using PSSTs has allowed not only to define semantics for regexes with capture groups, but also to define semantics for API functions like `replace` or `replaceAll`, albeit without support for lookarounds and backreferences. In contrast, our mechanization does not cover API functions (see section 7), but covers a wider range of regex constructs.

The semantics proposed in [Chida and Terauchi 2022] supports a combination of capture groups, lookaheads, and backreferences to prove theorems about the expressive power of combining these features. Interestingly, this semantics diverges from JavaScript on the interaction of lookaheads and backreferences. In JavaScript, lookaheads are *atomic* (see [ECMA 2023, 22.2.2.4, Note 3]), meaning that if there are several ways to match a lookahead, only the first one will be tried and a backreference cannot cause the engine to backtrack inside a lookahead.

Regex semantics. Differences between regex languages have been studied in various contexts. A number of syntactic and semantic differences have been documented in [Davis et al. 2019] for JavaScript, Java, PHP, Python, Ruby, Go, Perl and Rust regexes, with JavaScript being different from all others in several cases. That study already presented the JavaScript peculiarities of section 2.1.1, but not the first, third and fourth counterexamples of Figure 21. Recently, work that presented algorithms for matching a large subset of JavaScript regexes in linear time [Barrière and Pit-Claudel 2024] (albeit without mechanized proofs or a formal model) further showed that subtleties of quantifier semantics had caused bugs in V8. The semantics of backreferences are another source of differences across languages. Backreferences can either match the empty string or nothing when the corresponding group is undefined, and some languages can allow multiple definitions of named groups with the same name. Recent work [Berglund and van der Merwe 2017b] showed the impact of these semantic choices on the expressivity of the language. JavaScript backreferences correspond to their *No-label repetitions, ε-semantics* category.

9 Conclusion

We have presented an executable, proven-safe, faithful and future-proof Coq mechanization of JavaScript regex semantics. We have shown that with the right combination of encoding techniques, a manual shallow embedding of the ECMAScript backtracking algorithm is possible in Coq and can scale to the full specification. We have additionally proved that the specification always terminate and that failures never occur, two properties that could not have been formally verified without a mechanization. Our work can be used to conduct more proofs about JavaScript regex semantics, for instance proving the correctness of optimizations used in widely deployed implementations. Finally, our development can be extracted to OCaml or JavaScript, providing two executable regex engines. We validated the correctness of our mechanization with the Test262 official conformance test suite, and cross-validated it against the V8 Irregexp engine. To our knowledge, this is the first time the ECMAScript specification of regexes has been mechanized in a proof assistant.

Our work lays the foundations for future formal work about real-world regex languages. In particular, we hope that it will make it possible to prove the correctness of efficient matching algorithms, as well as rewriting-based optimizations for regexes. We also hope that it will provide a foundation for researchers to restate the semantics in a way that better suits their field, without sacrificing the assurance that these new formulations correctly reflect the ECMAScript regex language.

Acknowledgments

The authors would like to thank Steve Zdancewic, as well as the anonymous reviewers, for their suggestions and feedback.

This project was supported by the Open Research Data Program of the ETH Board.

Artifact Availability

The mechanization and proofs described in this paper are free software. They can be found online at https://github.com/epfl-systemf/Warblre.

A peer-reviewed artifact [De Santo et al. 2024a] is also available. It consists of a virtual machine with our Coq mechanization, our proof scripts, and our auxiliary code (fuzzer, tests), as well as scripts to recreate the virtual machine from scratch.

References

Aurèle Barrière and Clément Pit-Claudel. 2024. Linear Matching of JavaScript Regular Expressions. *Proceedings of the ACM on Programming Languages* 8, PLDI, Article 201 (June 2024), 25 pages. https://doi.org/10.1145/3656431

Martin Berglund, Frank Drewes, and Brink van der Merwe. 2014. Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching. In *Proceedings 14th International Conference on Automata and Formal Languages, AFL 2014, Szeged, Hungary, May 27-29, 2014 (EPTCS, Vol. 151)*, Zoltán Ésik and Zoltán Fülöp (Eds.). 109–123. https://doi.org/10.4204/EPTCS.151.7

Martin Berglund and Brink van der Merwe. 2017a. On the semantics of regular expression parsing in the wild. *Theor. Comput. Sci.* 679 (2017), 69–82. https://doi.org/10.1016/J.TCS.2016.09.006

Martin Berglund and Brink van der Merwe. 2017b. Regular Expressions with Backreferences Re-examined. In *Proceedings of the Prague Stringology Conference 2017, Prague, Czech Republic, August 28-30, 2017*, Jan Holub and Jan Zdárek (Eds.). Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 30–41. http://www.stringology.org/event/2017/p04.html

Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 87–100. https://doi.org/10.1145/2535838.2535876

Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *Journal of the ACM* 11, 4 (oct 1964), 481–494. https://doi.org/10.1145/321239.321249

Mathias Bynens. 2024. Tests for RegExp Unicode property escapes. https://github.com/mathiasbynens/unicode-property-escapes-tests.

Arthur Charguéraud. 2013. Pretty-Big-Step Semantics. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 41–60. https://doi.org/10.1007/978-3-642-37036-6_3

Arthur Charguéraud, Alan Schmitt, and Thomas Wood. 2018. JSExplain: A Double Debugger for JavaScript. In *Companion Proceedings of the The Web Conference 2018* (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 691–699. https://doi.org/10.1145/3184558.3185969

Taolue Chen, Alejandro Flores-Lamas, Matthew Hague, Zhilei Han, Denghang Hu, Shuanglong Kan, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. 2022. Solving string constraints with Regex-dependent functions through transducers with priorities and variables. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498707

Nariyoshi Chida and Tachio Terauchi. 2022. On Lookaheads in Regular Expressions with Backreferences. In *7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, August 2-5, 2022, Haifa, Israel (LIPIcs, Vol. 228)*, Amy P. Felty (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:18. https://doi.org/10.4230/LIPICS.FSCD.2022.15

Nariyoshi Chida and Tachio Terauchi. 2023. Repairing Regular Expressions for Extraction. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1633–1656. https://doi.org/10.1145/3591287

James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 246–256. https://doi.org/10.1145/3236024.3236027

James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2019. Why Aren't Regular Expressions a Lingua Franca? An Empirical Study on the Re-use and Portability of Regular Expressions. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 443–454. https://doi.org/10.1145/3338906.3338909

James C. Davis, Francisco Servant, and Dongyoon Lee. 2021. Using Selective Memoization to Defeat Regular Expression Denial of Service (ReDoS). In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1–17. https://doi.org/10.1109/SP40001.2021.00032

Noé De Santo, Aurèle Barrière, and Clément Pit-Claudel. 2024a. *Artifact for "A Coq Mechanization of JavaScript Regular Expression Semantics" at ICFP 2024 (latest)*. https://doi.org/10.5281/zenodo.11494316

Noé De Santo, Aurèle Barrière, and Clément Pit-Claudel. 2024b. *Artifact for "A Coq Mechanization of JavaScript Regular Expression Semantics" at ICFP 2024 (v1)*. https://doi.org/10.5281/zenodo.11494317

Mark Jason Dominus. 2000. Perl Regular Expression Matching is NP-Hard. https://perl.plover.com/NPC/NPC-3SAT.html.

ECMA. 2023. ECMA-262, 14th edition: ECMAScript® 2023 Language Specification. https://262.ecma-international.org/14.0/index.html

ESLint. 2024. regexpp. https://github.com/eslint-community/regexpp.

Arjun Guha and Benjamin Lerner. 2012. Mechanized LambdaJS. https://github.com/brownplt/LambdaJS/blob/master/coq/LambdaJS_Defs.v.

Gérard Huet. 1997. The zipper. *Journal of functional programming* 7, 5 (1997), 549–554.

Iain Ireland. 2020. A New RegExp Engine in SpiderMonkey. https://hacks.mozilla.org/2020/06/a-new-regexp-engine-in-spidermonkey/.

Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 5028)*, Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe (Eds.). Springer, 359–369. https://doi.org/10.1007/978-3-540-69407-6_39

Blake Loring, Duncan Mitchell, and Johannes Kinder. 2019. Sound Regular Expression Semantics for Dynamic Symbolic Execution of JavaScript. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and*

Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 425–438. https://doi.org/10.1145/3314221.3314645

Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2008. An Operational Semantics for JavaScript. In *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5356)*, G. Ramalingam (Ed.). Springer, 307–325. https://doi.org/10.1007/978-3-540-89330-1_22

Melange. 2024. Melange, OCaml for JavaScript developers. https://melange.re.

Takayuki Miyazaki and Yasuhiko Minamide. 2019. Derivatives of Regular Expressions with Lookahead. *Journal of Information Processing* 27 (2019), 422–430. https://doi.org/10.2197/IPSJJIP.27.422

Dan Moseley, Mario Nishio, Jose Perez Rodriguez, Olli Saarikivi, Stephen Toub, Margus Veanes, Tiki Wan, and Eric Xu. 2023. Derivative Based Nonbacktracking Real-World Regex Matching with Backtracking Semantics. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1026–1049. https://doi.org/10.1145/3591262

Scott Owens, John Reppy, and Aaron Turon. 2009. Regular-expression derivatives re-examined. *Journal of Functional Programming* 19, 2 (2009), 173–190.

Daejun Park, Andrei Stefanescu, and Grigore Rosu. 2015. KJS: a complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 346–356. https://doi.org/10.1145/2737924.2737991

Jihyeok Park, Seungmin An, and Sukyoung Ryu. 2022. Automatically deriving JavaScript static analyzers from specifications using Meta-level static analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 1022–1034. https://doi.org/10.1145/3540250.3549097

Jihyeok Park, Seungmin An, Wonho Shin, Yusung Sim, and Sukyoung Ryu. 2021a. JSTAR: JavaScript Specification Type Analyzer using Refinement. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 606–616. https://doi.org/10.1109/ASE51524.2021.9678781

Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu. 2021b. JEST: N+1 -version Differential Testing of Both JavaScript Engines and Specification. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 13–24. https://doi.org/10.1109/ICSE43902.2021.00015

Jihyeok Park, Jihee Park, Seungmin An, and Sukyoung Ryu. 2020. JISET: JavaScript IR-based Semantics Extraction Toolchain. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 647–658. https://doi.org/10.1145/3324884.3416632

Yuto Sakuma, Yasuhiko Minamide, and Andrei Voronkov. 2012. Translating regular expression matching into transducers. *Journal of Applied Logic* 10, 1 (2012), 32–51. https://doi.org/10.1016/J.JAL.2011.11.003

TC39. 2022. TC39 Meeting Notes. https://github.com/tc39/notes/blob/main/meetings/2022-01/jan-26.md.

TC39. 2023. Test262, commit `4d44acbc033c8aa38896f41fb3bdc96d6aeee2b8`. https://github.com/tc39/test262/commit/4d44acbc

V8. 2019. Improving V8 regular expressions. https://v8.dev/blog/regexp-tier-up.

V8. 2021. An Additional Non-backtracking RegExp Engine. https://v8.dev/blog/non-backtracking-regexp.

V8. 2023. Issue 14098: [Regexp] Mismatch between the Experimental engine and the backtracking engine on empty repetitions. https://bugs.chromium.org/p/v8/issues/detail?id=14098.

V8. 2024. The V8 JavaScript engine. https://v8.dev/

Ekaterina Zhuchko, Margus Veanes, and Gabriel Ebner. 2024. Lean Formalization of Extended Regular Expression Matching with Lookarounds. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024, London, UK, January 15-16, 2024*, Amin Timany, Dmitriy Traytel, Brigitte Pientka, and Sandrine Blazy (Eds.). ACM, 118–131. https://doi.org/10.1145/3636501.3636959

Received 2024-02-28; accepted 2024-06-18